ARTIFICIAL INTELLIGENCE

Report Lab: Gem Hunter

Teachers: Nguyễn Tiến Huy
Nguyễn Thanh Tình
Nguyễn Trần Duy Minh

Student:

MSSV	Name
23127259	Nguyễn Tấn Thắng

Thành phố Hồ Chí Minh, tháng 3 năm 2025

Contents

1 Purpose

- Develop a "Gem Hunter" game that uses Conjunctive Normal Form (CNF) for logical problem-solving.
  - The game involves players exploring a grid to locate hidden gems while avoiding traps.
  - Each numbered tile in the grid indicates the number of surrounding traps (ranging from 1 to 8), considering all eight adjacent cells (including diagonals).
  - The problem is formulated as a set of CNF constraints and solved using logical reasoning techniques.
- To address this problem, the following steps were undertaken:
  - Defined logical variables to represent the state of each cell in the grid (trap or gem).
  - Developed a modular codebase to handle various aspects of the game, including:
    - `grid.py`: Manages the grid structure and provides utility functions for accessing cells and their neighbors.
    - `constraints.py`: Defines and generates CNF constraints based on the grid's rules.
    - `pysat_solver.py`: Solves the CNF constraints using the PySAT library with the Glucose3 solver.
    - `brute_force_solver.py`: Implements a brute-force approach to solve the puzzle for comparison.
    - `backtracking_solver.py`: Implements a backtracking algorithm to solve the puzzle for comparison.
    - `main.py`: Orchestrates the game by reading input, invoking solvers, and writing output.
  - Evaluated the implementation using test cases to validate the correctness and performance of the solvers. The test cases include:
    - `input_1.txt`: A 5x5 matrix.
    - `input_2.txt`: A 11x11 matrix.
    - `input_3.txt`: A 20x20 matrix.

2 Solution description: Describe the correct logical principles for generating CNFs

- The "Gem Hunter" game is modeled as a satisfiability (SAT) problem using Conjunctive Normal Form (CNF) constraints, ensuring each numbered tile has the exact number of traps in its surrounding cells (up to 8 neighbors, including diagonals).

  - Convert the game into a SAT problem by creating CNF constraints.
    * Ensure each numbered tile reflects the correct count of nearby traps.
    * Solve by defining variables, building constraints, and applying a SAT solver.
  - Key principles for generating CNF constraints:
    * Assign variables to the grid: each empty cell ('_') gets a Boolean variable ('True' for trap 'T', 'False' for gem 'G'); fixed cells ('T' or 'G') are skipped. In 'constraints.py', variables are indexed as $r \times C + c + 1$, with R rows, C columns, and (r, c) as the cell position.
    * Set constraints for pre-filled cells: add '[var]' for a trap ('T') to enforce 'True', or '[-var]' for a gem ('G') to enforce 'False', preserving the grid's initial configuration.
    * Create constraints for numbered cells: for a cell (r, c) with value N, ensure N traps in its 8 surrounding cells. Find neighbors, count empty ones (size n), and pre-assigned traps (t). Compute traps needed: $k = N - t$. Address edge cases: if $k < 0$ (excess traps) or $k > n$ (insufficient empty cells), the grid has no solution; if $k = 0$, set empty neighbors to gems with '[-var]'; if $k = n$, set them to traps with '[var]'. For $0 < k < n$, apply at-most constraint $U(k, n)$ (in any $k+1$ empty neighbors, at least one is not a trap) and at-least constraint $L(k, n)$ (in any $n - k + 1$ empty neighbors, at least one is a trap), ensuring exactly k traps.
    * Resolve the CNF constraints: use PySAT (with Glucose3 solver) to find a valid assignment, then map results to the grid: 'True' as 'T', 'False' as 'G'.

3 Generate CNFs Automatically

- Automatically generate Conjunctive Normal Form (CNF) constraints for the "Gem Hunter" game.
  - In `constraints.py`, the program creates CNF clauses based on the grid's rules.
    * Assign Boolean variables to empty cells ('True' for traps 'T', 'False' for gems 'G').
    * Add constraints for pre-filled cells: $[var]$ for 'T', $[-var]$ for 'G'.
    * For numbered cells with value N, compute traps needed ($k = N - t$), where t is the number of pre-assigned traps, and ensure k traps in empty neighbors using at-most and at-least constraints.
  - The generated CNF clauses are then solved using PySAT to find a valid grid configuration.

- **Using Mathematical Logic and Clauses:**
  - Assign variables to each cell, where $x_{i,j}$ is a Boolean variable representing the state of cell (i, j):
    - $x_{i,j}$ = True if the cell contains a trap ('T').
    - $x_{i,j}$ = False if the cell contains a gem ('G').
  - **Pre-filled cells**: If a cell is pre-filled with a trap or a gem, we add the corresponding CNF clause:

$$[x_{i,j}] \quad \text{for trap 'T'} \quad \text{and} \quad [-x_{i,j}] \quad \text{for gem 'G'}.$$

  - **For numbered cells with value** N: A numbered cell represents the number of traps around it. For a given numbered cell (i, j), we know the number of traps needed $k = N - t$, where t is the number of already assigned traps in neighboring cells. Thus, we need to ensure that exactly k of the neighboring cells contain traps.
  - **At-most and at-least constraints**: These constraints ensure that exactly k traps are placed among the empty neighboring cells. Let the variables representing the empty neighboring cells of (i, j) be denoted as:

$$\{x_{i-1,j-1}, x_{i-1,j}, x_{i-1,j+1}, x_{i,j-1}, x_{i,j+1}, x_{i+1,j-1}, x_{i+1,j}, x_{i+1,j+1}\}.$$

  - To ensure that exactly k of these variables are true (representing traps), we generate a combination of at-most and at-least clauses. Specifically:
    - **At-most constraints**: We ensure that no more than k traps are placed in these neighboring cells. This is done using the following set of clauses:

$$\bigwedge_{i=1}^{\text{(number of combinations of k traps)}} \left(\bigvee_{j=1}^{k} x_{i,j} \right)$$

      where the sum of selected combinations will be at most k.
    - **At-least constraints**: We ensure that at least k traps are placed in the neighboring cells. The constraints are constructed by ensuring that at least k variables among the neighboring cells must be true:

$$\bigwedge_{i=1}^{k} \left(\bigwedge_{j=1}^{\text{(minimum number of traps)}} x_{i,j} \right)$$

  - **Final CNF clauses**: All the CNF clauses are combined to form the complete CNF representation of the puzzle. The clauses are then solved using PySAT or brute force or backtracking.

4 Source code

4.1 File grid.py

- This file defines a class Grid that is used to manage a matrix and provides methods to interact with it. The grid can store data in a 2D array and includes functionalities to retrieve cell values, identify neighboring cells, and print the matrix in a user-friendly format. The class can be used to perform operations related to matrix manipulation and querying its structure.
- In this file, it can be seen that:
  - Function __init__ is a constructor that initializes the matrix with the given data and calculates the number of rows and columns.
  - Function get_cell is a method that:
    - Retrieves the value of a cell at a specific row and column.
    - Returns None if the given position is out of bounds.
  - Function get_neighbors is a method that:
    - Retrieves the neighboring cells of a given cell (including diagonal neighbors).
    - Returns a list of tuples containing the row, column, and value of each neighboring cell.
  - Function print_grid is a method that Prints the matrix in a human-readable format, replacing None values with an underscore (_) and separating cells with commas.

4.2 File constraints.py

- This file defines a class Constraints that is responsible for generating logic constraints in CNF (Conjunctive Normal Form) for a grid-based puzzle, where certain cells may represent traps ('T'), gems ('G'), or numbers that specify how many traps are adjacent. The class uses SAT (Satisfiability) solving techniques to model the puzzle and generate the appropriate CNF clauses to solve the problem. The constraints are designed to ensure that the puzzle rules are satisfied, such as:
  - Trap cells are marked correctly.
  - Gem cells are marked as not containing traps.
  - The number of traps in adjacent cells must meet the number specified in numbered cells.
- In this file, it can be seen that:

- Function __init__ is a constructor that initializes the grid, number of rows and columns, a CNF object to store clauses, a variable map for associating cells with variables, and a counter for the next variable to assign.
- Function assign_variables is a method that assigns variables to empty cells (those with no pre-assigned value). These variables represent whether a cell is a trap (T) or not.
- Function add_pre_assigned is a method that adds constraints for cells that are pre-assigned with traps ('T') or gems ('G'). It updates the CNF to reflect these assignments.
- Function add_number_constraints is a method that:
  - Adds constraints based on the numbers in the grid, specifying how many adjacent cells must contain traps. It ensures that the number of traps placed in neighboring cells is consistent with the number given in each numbered cell.
  - This includes checking if the puzzle is solvable, handling edge cases where constraints are not feasible (e.g., too many traps are already placed).
- Function generate_cnf is a method that:
  - Combines the above methods to generate the CNF clauses for the entire grid.
  - Returns the CNF clauses (representing the puzzle constraints) and a variable map.

4.3 File pysat_solver.py

- This file defines a class PySATSolver which is responsible for solving the grid-based puzzle problem using the PySAT library, specifically the Glucose3 SAT solver. The solver takes a CNF (Conjunctive Normal Form) representation of the puzzle and attempts to find a solution by applying logical constraints. It returns the solution (a matrix with traps 'T' and gems 'G') and the time taken to solve the problem.
- In this file, it can be seen that:
  - Function __init__ is a constructor that:
    - Initializes the grid, CNF clauses, and variable map.
    - Initializes the solution as None, which will later hold the solved puzzle grid.
  - Function solve is a method that:
    - Solves the puzzle using the Glucose3 SAT solver.
    - Adds all CNF clauses to the solver.

    - If the CNF is empty, assigns 'G' (Gem) to all empty cells and returns a solution.
    - Otherwise, attempts to solve the CNF using the solver. If a solution is found, updates the puzzle grid with traps ('T') and gems ('G') based on the solver's model.
  - Function print_solution is a method that prints the solution to the console if one exists. If no solution is found, it outputs a message indicating that no solution was found.

4.4 File brute_force_solver.py

- The BruteForceSolver class is designed to solve the grid-based puzzle problem using a brute force approach. It systematically checks all possible combinations of trap ('T') and gem ('G') assignments for the empty cells in the grid. For each combination, it checks if the resulting grid satisfies the constraints (i.e., the number of traps adjacent to each numbered cell). If a valid solution is found, it returns the solution and the time taken to find it.

- In this file, it can be seen that:

  - Function __init__ is a constructor that initializes the grid and sets the initial solution as None. ‘
  - Function solve is a method that:
    - Solves the puzzle by iterating over all possible combinations of trap ('T') and gem ('G') assignments for the empty cells.
    - For each combination, it constructs a new grid and checks if it satisfies the constraints using the check_grid method.
    - If a valid grid is found, it stores the solution and stops the search.
    - Returns whether a solution was found and the time taken to find it.
  - Function check_grid is a method that:
    - Verifies if a given grid satisfies the constraints for all numbered cells.
    - For each numbered cell, it checks if the number of adjacent traps matches the number specified in the cell.
  - Function print_solution is a method that prints the solution to the console if one exists. If no solution is found, it outputs a message indicating that no solution was found using brute force.

4.5 File backtracking_solver.py

- The BacktrackingSolver class is designed to solve the grid-based puzzle using the backtracking algorithm. It attempts to assign values ('T' for trap,

'G' for gem) to the empty cells in the grid and uses the backtracking technique to recursively explore possible configurations. If a valid configuration is found that satisfies the puzzle constraints, it returns the solution. If no solution is found, it backtracks to explore other possibilities. The class also tracks the time taken to solve the puzzle.

- In this file, it can be seen that:
  - Function __init__ is a constructor that initializes the grid and sets the initial solution as None.
  - Function solve is a method that:
    - Solves the puzzle using the backtracking approach.
    - Creates a working grid (a copy of the original grid) to be modified during the solving process.
    - Calls the backtrack method to try to find a valid configuration.
    - Returns whether a solution was found and the time taken to solve the puzzle.
  - Function backtrack is a method that:
    - The core backtracking method that recursively assigns values ('T' or 'G') to empty cells in the grid.
    - It checks if the current partial configuration is valid using the is_partial_grid_valid method.
    - If a valid configuration is found, it proceeds to the next cell; otherwise, it backtracks.
    - If the grid reaches the end and all constraints are satisfied, it returns the solved grid.
  - Function is_partial_grid_valid is a method that:
    - Checks if the current partial configuration (up to a certain cell) is valid.
    - For each numbered cell, it checks if the number of adjacent traps is within the acceptable range based on the current configuration.
  - Function check_grid is a method that:
    - Verifies if the complete grid satisfies all the puzzle constraints.
    - For each numbered cell, it checks if the number of adjacent traps matches the number specified in the cell.
  - Function print_solution is a method that prints the solution to the console if one exists. If no solution is found, it outputs a message indicating that no solution was found using backtracking.

4.6 File main.py

- This file is a solver program for a grid-based puzzle. It uses three different solving techniques—PySAT, Brute Force, and Backtracking—to solve the

puzzle and generates solutions using each method. The program reads input from text files, processes the grid data, applies the solving algorithms, and writes the results to output files. Additionally, it offers a user interface to select which puzzle to solve and provides a summary of results for each method.

- In this file, it can be seen that:
  - Function read_matrix_from_file is a function reads a matrix (grid data) from a text file. It parses the data and returns it as a list of lists. Cells with (_) are considered empty, and numbers are parsed as integers while other values (like 'T' and 'G') are treated as strings.
  - Function write_all_solutions_to_file is a function writes all the solutions (from different solving methods) to a specified output file. It formats the solutions clearly, indicating the solver used for each result.
  - Function solve_matrix is a function solves the puzzle by:
    - Creating a Grid object from the provided matrix.
    - Generating the CNF constraints for solving with PySAT.
    - Applying each solver (PySAT, Brute Force, and Backtracking).
    - Printing the results and recording the time taken for each solution.
    - Writing all solutions to an output file.
  - Function display_menu is a function displays a menu to the user to choose which input file to process. The user selects the file to solve, and the program proceeds with solving that specific puzzle.
  - Function main works:
    - Checks for the existence of required input files (input_1.txt, input_2.txt, input_3.txt).
    - Reads the matrices from those files.
    - Provides a menu for the user to choose which file to solve.
    - Calls the solve_matrix function to solve the selected puzzle and output the results.

5 Results

- To begin with the test case in file `input_1.txt` that holds a matrix of size 5x5:
  - The original matrix is not solved:

```
Initial map
_, 2, 2, 2, 1
_, _, _, 2, _
_, 3, _, _, 2
_, _, 2, 3, _
_, 2, _, _, _
```

Figure 1: The original matrix 5x5 is not solved

– The result on terminal using the PySAT library:

```
Solving with PySAT...

Result map
G, 2, 2, 2, 1
G, T, T, 2, T
G, 3, G, G, 2
T, G, 2, 3, T
G, 2, T, T, G
Running time: 0.00501 seconds
Có sử dụng solver của pysat
```

Figure 2: The result using PySAT library

– The result on terminal using brute force:

```
Solving with Brute Force...

Result map
G, 2, 2, 2, 1
G, T, T, 2, T
T, 3, G, G, 2
G, G, 2, 3, T
T, 2, T, T, G
Running time: 0.53125 seconds
Sử dụng thuật toán bruteforce
```

Figure 3: The result using brute force

– The result on terminal using backtracking:

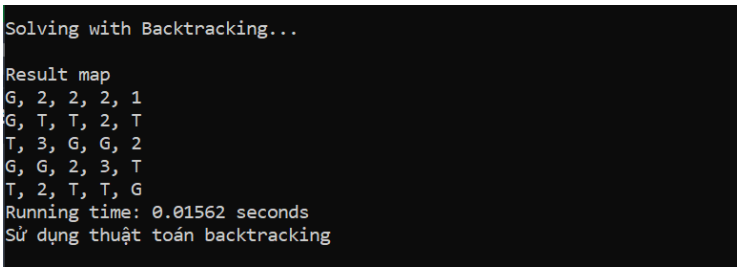

Figure 4: The result using backtracking

- To continue with the test case in file input_2.txt that holds a matrix of size 11x11:
  - The original matrix 11x11 is not solved:

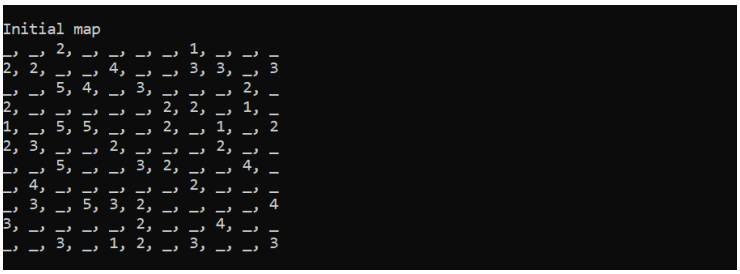

Figure 5: The original matrix is not solved

  - The result on terminal using the PySAT library:

Figure 6: The result using PySAT library

  - The result on the terminal using brute force: Due to the complex structure of this matrix and the large number of grid checks required, the algorithm's complexity is $O(2^n)$, leading to slow execution and potentially failing to return a result.

– The result on terminal using backtracking:

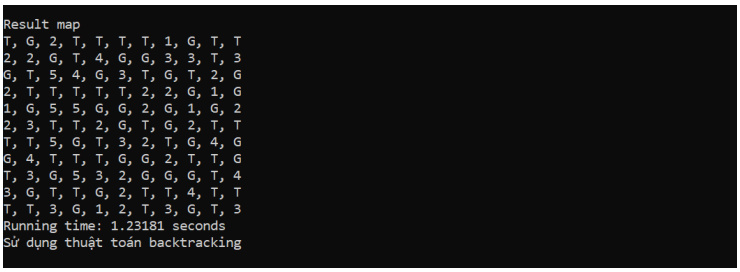

Figure 7: The result using backtracking

- To continue with the test case in file input_3.txt that holds a matrix of size 20x20:

  – The original matrix 20x20 is not solved:

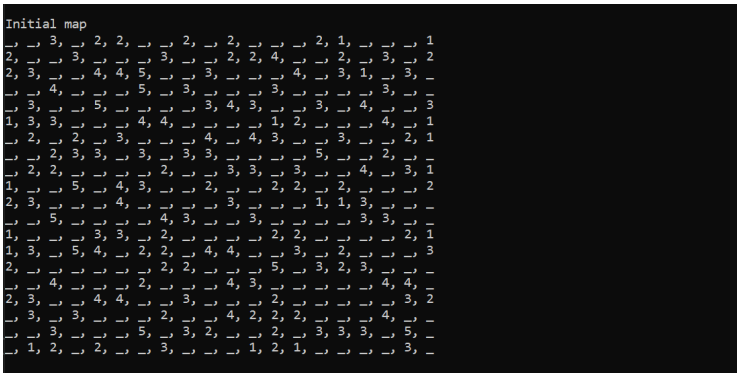

Figure 8: The original matrix is not solved

– The result on terminal using the PySAT library:

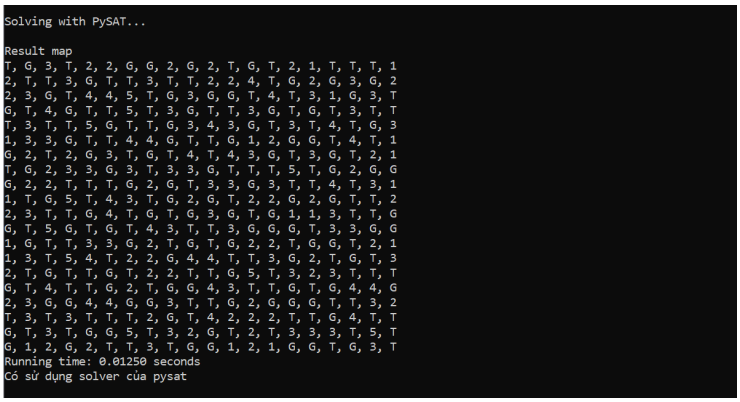

Figure 9: The result using PySAT library

– The result on the terminal using brute force: Due to the complex structure of this matrix and the large number of grid checks required, the algorithm's complexity is $O(2^n)$, leading to slow execution and potentially failing to return a result.

– The result on terminal using backtracking:

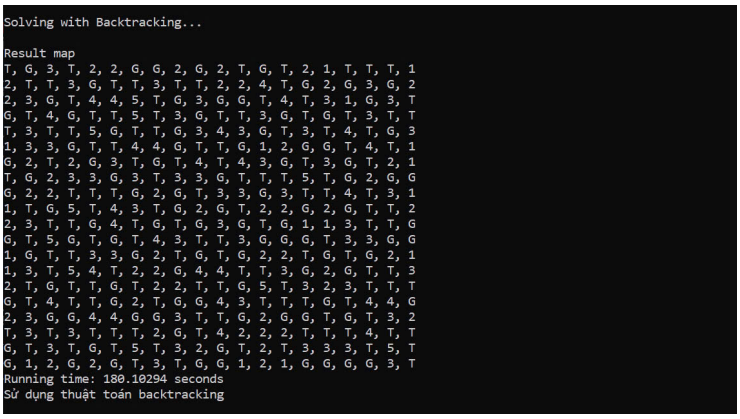

Figure 10: The result using backtracking

6 Compare program brute-force algorithm with using library(speed)

- In the results in figure 2 and figure 3, we can see that: the running time using the PySAT library is significantly less than the running time using the brute-force approach. The running time using the PySAT library

is 0.00501 seconds, while the running time using brute force is 0.53125 seconds.

- This can be explained replied on time complexity and the structures of matrix:
  - Pysat libarary:
    - The PySAT solver uses a SAT-solving approach to determine the satisfiability of a set of logical constraints. It converts the problem into Conjunctive Normal Form (CNF) and then applies highly optimized algorithms to solve the SAT problem.
      - The time complexity of SAT solvers like PySAT depends on the structure of the problem and the solver's internal heuristics, but in general, it can be much more efficient than brute force.
      - The typical time complexity of a SAT solver is exponential in the worst case, but in practice, it often performs much better due to the use of conflict-driven learning, clause propagation, and other optimization techniques. The average time complexity can vary depending on how easily the problem can be reduced, but in practice, it is much faster than brute-force for problems with non-trivial constraints.
    - PySAT uses more advanced algorithms to improve performance. This results in a significant reduction in the solving time for large problems, which is why the running time using PySAT is far smaller than that of brute force.
  - Brute force algorithm:
    - The brute-force algorithm generates all possible combinations of values for the empty cells in the grid and then checks whether each combination satisfies the puzzle's constraints.
      - For a grid of size $m \times n$, the number of empty cells can be as high as $m \times n$.
      - Each empty cell can have 2 possible values (e.g., 'T' or 'G'), leading to $2^{m \times n}$ possible configurations to check.
      - Thus, the time complexity of the brute-force algorithm is $O(2^{m \times n})$, which grows exponentially as the grid size increases.
    - As seen in our comparison, the brute-force method becomes slower for larger grids, as it needs to explore all possible combinations. This exponential time complexity makes brute-force impractical for large or even medium-sized puzzles.
- In conclusion, from the results in figure 2 and figure 3, it is clear that for problems of even moderate size, using a library like PySAT significantly reduces the running time compared to the brute-force approach. The PySAT method performs well even for larger and more complex grids

due to its efficiency, whereas brute force quickly becomes impractical for larger problems. In general, when solving large or complex puzzles with constraints, the **PySAT library** is a far superior method in terms of both time efficiency and practicality compared to the brute-force algorithm.

7 Compare program backtracking algorithm with Using Library (Speed)

- In the results in figure 2 and figure 4, we can see that: the running time using the PySAT library is significantly less than the running time using the backtracking approach. The running time using the PySAT library is 0.00501 seconds, while the running time using backtracking is 0.01562 seconds.
- This can be explained based on time complexity and the structures of the matrix:
  - PySAT Library:
    - The PySAT solver uses a SAT-solving approach to determine the satisfiability of a set of logical constraints. It converts the problem into Conjunctive Normal Form (CNF) and then applies highly optimized algorithms to solve the SAT problem.
      - The time complexity of SAT solvers like PySAT depends on the structure of the problem and the solver's internal heuristics, but in general, it can be much more efficient than backtracking.
      - The typical time complexity of a SAT solver is exponential in the worst case, but in practice, it often performs much better due to the use of conflict-driven learning, clause propagation, and other optimization techniques. The average time complexity can vary depending on how easily the problem can be reduced, but in practice, it is much faster than backtracking for problems with non-trivial constraints.
    - PySAT uses more advanced algorithms to improve performance. This results in a significant reduction in the solving time for large problems, which is why the running time using PySAT is far smaller than that of backtracking.
  - Backtracking Algorithm:
    - The backtracking algorithm solves the problem by trying all possible values for empty cells, starting from the first empty cell and recursively filling the grid. If a valid solution is found, the algorithm continues; otherwise, it backtracks and tries other possibilities.

- The time complexity of backtracking depends on the number of empty cells and the branching factor (number of choices for each cell). For a grid of size $m \times n$, the branching factor is 2 (for 'T' or 'G' in each cell).
- In the worst case, backtracking explores all possible combinations of empty cells, leading to a time complexity of $O(2^{m \times n})$, similar to the brute-force algorithm.
- In practice, backtracking often performs better than brute force because it prunes invalid branches early, but the overall time complexity remains exponential.

* As seen in our comparison, backtracking becomes slower for larger grids as it needs to explore many possible combinations and backtrack when an invalid path is found. While backtracking is more efficient than brute-force in some cases, its exponential time complexity makes it impractical for large problems.

- In conclusion, from the results in figure 2 and figure 4, it is clear that for problems of even moderate size, using a library like PySAT significantly reduces the running time compared to the backtracking approach. The PySAT method performs well even for larger and more complex grids due to its efficiency, while backtracking, though faster than brute force, still suffers from exponential time complexity. In general, when solving large or complex puzzles with constraints, the **PySAT library** is a far superior method in terms of both time efficiency and practicality compared to the backtracking algorithm.

8 Reference Material

- The following sources were referenced during the development of this project:
  - Link: `https://github.com/Tuprott991/Artificial-intelligence-basis-Gem-Hunter`
  - ChatGPT

9 Video demo

- Link: `https://www.youtube.com/watch?v=7mNUZJUCVcM`